

WHAT'S NEW IN POSTS FOR X8

This document describes the key new features related to MP post development for Mastercam X8.

Where to get updated documentation—to get the latest MP documentation, download version 1.90 or higher of the **MP_documentation.pdf** [from our extranet](#).

Note that new/changed/removed parameters are not discussed in this document. These are fully documented in the complete *NCI & Parameter Reference* document found in the **MP_documentation.pdf**.

New and improved functions

Several functions have been exposed to post developers or enhanced.

Changes to `launch()` function

Use the `launch()` function to launch and run another program from your post. For X8, it has been enhanced with a third parameter that lets you specify whether the new program will be run as a modal dialog. The program to be run needs to be an executable file with an **.exe** extension. (Use the `dll()` function to run a **.dll** executable.)

General form

```
return = launch(progname, argument, mode)
```

where:

- **return** = an integer variable to store the return value. The function returns 0 if successful, otherwise -1.
- **progname** = the name of the program that you want to run, including the extension. This should be enclosed within quotes. It should include the full path if it is not located in the same \apps folder as mp.dll.
- **argument** = an optional string that contains arguments to be passed to the program. Multiple arguments are typically enclosed in nested quote marks like this:

```
' "arg1" "arg2" "arg3" '
```

But the proper format for this depends on how the receiving program expects the arguments to be presented.

- **mode** = an optional argument that indicates whether the new program is to be opened as a modal dialog.
 - If this value is **0**, Mastercam will be locked and post processing will be suspended until the new program is closed.
 - If this value is **1**, Mastercam can be used while the new program is open.

The mode parameter is not supported in versions of Mastercam earlier than X8. If it is not included, the launch command will be processed as if mode **0** were specified.

Using nested quote marks

The name and path of the program to be run should be enclosed in quote marks if you are passing it as a literal string, but does not require nested quotes even if it contains spaces. The second argument—the list

of arguments to be passed to the destination program—might or might not require nested quotes. If the individual strings in the list of arguments contain spaces, use nested quotes as a best practice.

When nesting quotes, use double-quotes " " for the inner set, and single-quotes ' ' for the outer layer. If you are only using a single set of quotes, use double-quotes.

Using `launch()` to run an external program

This example launches an external program after post processing has been completed. It passes the new program the name and location of the NC file that has just been created by the post. The new program opens as a modeless dialog, so you can continue to use Mastercam while the new program is running.

```
# Define the full path\name of the EXE to be run
strEXE  "C:\Program Files (x86)\MyUtilities\myutility.exe"

# Create string with path\name of NC file
strfilename : spathnc$ + sname$ + sextnc$

ppost    # ppost is run AFTER all the files processed by MP are closed!
        result = launch(strEXE, strfilename, 1)
```

setncstr()

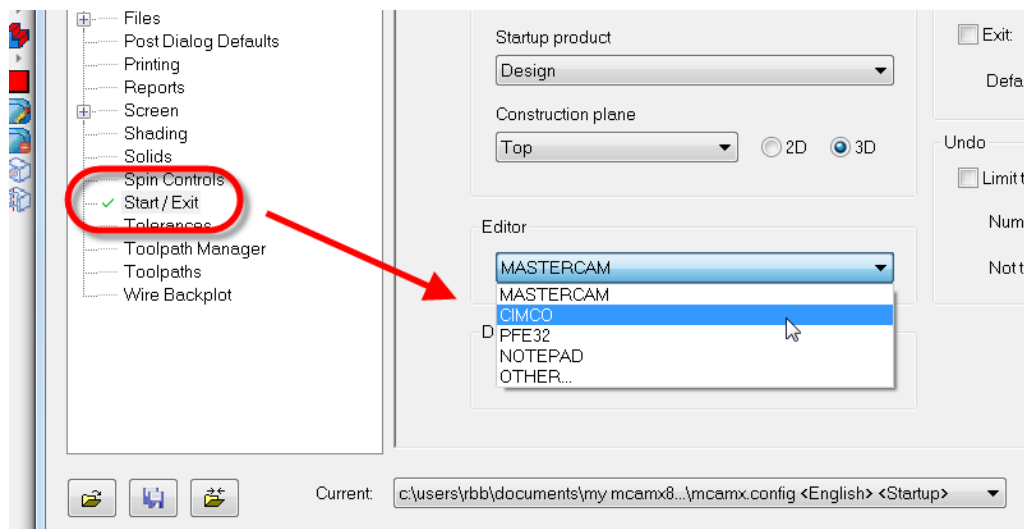
Use the `setncstr()` function to manage a list of files that is passed to the editor. Use it to build the list of files that the editor will open when posting is finished. This function is used most often when your NC output is distributed among more than one file, or your post needs to create several files, in a single posting session.

General form

```
return = setncstr(mode, filename, num)
```

where:

- `return` = an integer that indicates whether or not the function was successful: **1** if the function was successful, otherwise **0**.
- `mode` = an integer or integer variable that indicates the operating mode of the function. Valid values are **2**, **3**, and **4**.
 - A value of **2** indicates that the file indicated by `filename` is to be added to the list of files that will be opened by the editor.
 - A value of **3** indicates that the file indicated by `filename` is to be removed from the list of files that will be opened by the editor.
 - A value of **4** indicates that `filename` specifies the actual editor to be opened. This should be the full path and name of the editor's **.EXE** file. This overrides the editor that is selected in the user's configuration file:



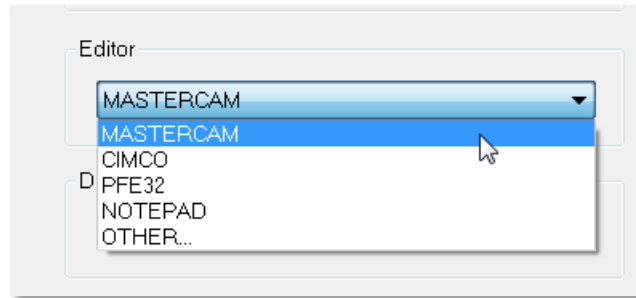
If the specified editor is not found (or the string is empty), then the editor from the configuration file is used. (This value is not supported in Mastercam X7, only X8 and later.)

- `filename` = a string that indicates the full path and filename as described in `mode`. If you are using a literal string, enclose it in double-quote marks (" ").
- `num` = a two-digit code that controls the editor's behavior. (This parameter is only valid when `mode` = **4**. It is not supported in Mastercam X7, only X8 and later.)
 - When the right (1's) digit is **0**, Mastercam will pause until the editor is closed. This matches the behavior of Mastercam versions before X7. If multiple files have been sent to the editor,

Mastercam will pause after each editor session. Use a value of **1** to leave Mastercam running and active along with the editor window.

- When the left (10's) digit is **0** (or omitted), Mastercam will create a separate instance of the editor for each file. Use this to guard against situations when an editor might not be able to accept multiple files. When this digit is **1**, Mastercam will supply the multiple-file list to the editor. This is appropriate for editors like Mastercam's Code Expert Editor, which support having multiple files open at the same time with each document in a separate tab.

Note that this behavior is “hard-wired” for the editor options that come from Mastercam's configuration file; in other words, the ones listed here:



- The **MASTERCAM** editor (in other words, the Code Expert Editor), **CIMCO**, and **PFE** options automatically behave like the left digit is **1**.
- The **NOTEPAD** option automatically behaves like the left digit is **0**.
- The editor that you select with **OTHER** will behave however you set the left-digit option in the post.

You can override the behavior of the standard editors by including a `setncstr()` statement in your post with `mode = 4`, and explicitly referencing the `.EXE` file for that editor. (But don't override the **NOTEPAD** option—Notepad will not work correctly if you do.)

Building a list of files for the editor

This example builds a list of NC files for each stream, then specifies the editor that will be used. Because the final parameter = **11**, all four files will open at the same time, with the Mastercam window and editor window both enabled.

Because of the extra parameter on the final line, this example will only work in Mastercam X8.

```
result t = setncstr(2, spathnc$ + snamenc$ + "_STREAM1" + sextnc$)
result t = setncstr(2, spathnc$ + snamenc$ + "_STREAM2" + sextnc$)
result t = setncstr(2, spathnc$ + snamenc$ + "_STREAM3" + sextnc$)
result t = setncstr(2, spathnc$ + snamenc$ + "_STREAM4" + sextnc$)
result t = setncstr(4, "C:\Program Files\MyEditor\ACME-Editor.exe", 11)
```

fstack

Use the `fstack` declaration to define a data stack in memory. This lets you define, create, and store records that can be accessed later.

- In a typical stack, new records are added to the top of the stack, pushing existing records down.
- When you read from the stack, you typically read the top record, and the remaining records are moved up.

Once you define the stack, use the `push()`, `pop()`, and `turn()` functions to add and retrieve records.

Records that are added to the stack are typically read from an implied array, where each variable in the array represents one field from the record. Similarly, when a record is read from the stack, it is typically stored in an implied array.

General form

```
fstack num1 num2
```

where:

- `num1` = the ID number of the stack
- `num2` = the number of fields in each record

Creating a data stack and manipulating it with `push()` and `pop()`

This example creates a data stack with two fields per record. It then uses the `push()` and `pop()` functions to add a record; query the stack size; and get a record.

```
# Define helper variables
stack_size : 0      # number of records in stack
result : 0          # used to report success/failure

# Define implied array
t_1 : 0             # first field in record
t_2 : 0             # second field in record

# Define stack
fstack 1 2          # define stack #1 with 2 fields per record

t_1 = t_1 + 1        # set values for t_1...
t_2 = t$             # ...and t_2

t_1 = push(1, result, 0) # create first record by pushing
                        # t_1 and t_2 onto the top of stack 1

stack_size = pop(1, result, 0) # query how many records are in stack 1

t_1 = pop(1, result, 1)    # get the top record of the stack...
                        # ... and store values in t_1 and t_2
```

push()

Use the `push()` function to add a record to an existing data stack. The stack must already have been created with the `fstack` command.

General form

```
val 1 = push(stack, option, mode)
```

where:

- `val 1` = the first variable in an implied array. Each variable represents a field in the record that will be added to the stack. MP will read the appropriate number of values from the implied array based on the record dimensions from the stack declaration.
- `stack` = the number of the stack.
- `option` = a numeric variable. For most modes, this will store the result of the operation: either **1** or **0** for success/failure in creating the record. However, some modes use this value differently as described below. Regardless, it is required that `option` be the name of a variable, not just a number.
- `mode` = the function mode. You can add the new record to the stack in any of several different ways, according to the value that you select for this parameter.

- **-1**—Read values from a buffer that has been previously defined with the `fbuf` statement. In this case, `option` indicates the number of the buffer from which to read the data. MP will iterate the buffer, empty the stack, and then copy the data from the buffer to the top of the stack. The success/fail code will be stored in `val 1`.

Note that because `fstack` does not support string data, the buffer must be defined as a real data buffer, not a string buffer.

- **0**—Push record (indicated by `val 1`) to the top of the stack. This is the default mode.
- **1**—Push record to the bottom of the stack.
- **2**—Push record below a specified record in the stack. In this case, `option` indicates the number of the record below which the new record will be inserted. (Note that the top record is record number 1; 0 is not a valid index value.)

Note that the success/fail code will be handled differently in this case. If you consider `option` to be the first element in an implied 2-element array, the success/fail code will be written to the second element. For example:

```
stack_index = 7
push_result = 999
val 1 = push(stack, stack_index, 2)
```

In this case, the result would be written to `push_result`, where `stack_index` and `push_result` together define a 2D array.

pop()

Use the **pop()** function to get data from a data stack. The stack must already have been created with the **fstack** command.

General form

val 1 = **pop(stack, option, mode)**

where:

- **val 1** = the first variable in an implied array. Each variable represents a field in the record that will be retrieved from the stack. As MP reads the record from the stack, each field will be stored in a variable in the array.
- **stack** = the number of the stack.
- **option** = a numeric variable. For most modes, this will store the result of the operation: either 1 or 0 for success/failure in getting the record. However, some modes use this value differently as described below. Regardless, it is required that **option** be the name of a variable, not just a number.
- **mode** = the function mode. You can query the stack in any of several different ways, according to the value that you select for this parameter.

- **-1**—Write the data record to a buffer that has been previously defined with the **fbuf** statement. In this case, **option** indicates the number of the buffer where the data will be stored. The success/fail code will be stored in **val 1**.

Note that because **fstack** does not support string data, the buffer must be defined as a real data buffer, not a string buffer.

- **0**—Return the number of records in the stack. The number will be stored in **val 1**.
- **1**—Get the top record from the stack, write it to the array, and remove it from the stack.
- **2**—Remove the top record from the stack without writing it to the array. In this case, both **val 1** and **option** will store the success/fail value.
- **3**—Get the top record from the stack, write it to the array, but leave it in the stack.
- **4**—Remove multiple records from the stack. The success/fail code will be stored in **val 1**.

In this case, the **option** value indicates the number of the first record to be deleted, and its sign indicates whether to delete the records before or after it.

If **option** is positive, **pop()** will delete that record and all records below it in the stack. For example, if there are 12 records in the stack, and the value of **option** is **5**, records 5–12 will be deleted.

If **option** is negative, **pop()** will delete that record and all records above it in the stack. For example, if there are 12 records in the stack, and the value of **option** is **-5**, records 1–5 will be deleted.

If the value of **option** is **-1**, **0**, or **1**, all the records in the stack will be deleted.

- **5**—Get a specific record from the stack, write it to the array, but leave it in the stack. In this case, **option** indicates the number of the record to get (the top record is record number 1). (Note that the top record is record number 1; 0 is not a valid index value.)

Note that the success/fail code will be handled differently in this case. If you consider `opt i on` to be the first element in an implied 2-element array, the success/fail code will be written to the second element. For example:

```
stack_index = 7
pop_result = 999
val 1 = pop(stack, stack_index, 5)
```

In this case, the result would be written to `pop_result`, where `stack_index` and `pop_result` together define a 2D array.

- The success/fail code will be written to the implied array in position `val 2`.
- **6**—Just like mode 5, but the record is removed from the stack.
- **7**—Get the bottom record from the stack, write it to the array, and remove it from the stack.

turn()

Use the `turn()` function to cycle the records in a stack. In other words, move the bottom record to the top and shift the other records down. You can also do the opposite: move the top record to the bottom, and shift the other records up. The stack must already have been created with the `fstack` command and contain at least two records.

General form

```
return = turn(stack, mode)
```

where:

- `return` = a variable that stores the result of the operation: either 1 or 0 for success/failure.
- `stack` = the number of the stack.
- `mode` = the function mode. There are two possible values:
 - **0**—Move bottom item to top and shift records down.
 - **1**—Move top item to bottom and shift records up.

fsg()

Mastercam X8 includes support for the `fsg()` function. Use the `fsg()` function to return a value based on a conditional expression. This can be a compact and efficient alternative to a regular boolean code block.

General form

`return = fsg(expression, x, y)`

where:

`return` = a variable to store the return value. This can be either a numeric variable or a string variable, but its type must be the same as `x` and `y`.

`expression` = a numeric variable, boolean expression, or other mathematical expression. The following examples are all valid expressions:

- `ipr_actv$`
- `(min_speed > 100)`
- `(state_one + state_two)`

`x` = the value that is returned if `expression` is true (or any non-zero value). This can be a number, the name of a numeric variable, or a mathematical expression.

`y` = the value that is returned if `expression` is false (or zero). This can be a number, the name of a numeric variable, or a mathematical expression.

`x` and `y` can also be strings, but they must be the same type: either both strings, or both numeric values.

If you use an expression for any of the arguments instead of a simple variable name, the recommended best practice is to enclose it in parentheses.

Getting the location of the user's shared folder

Recall that Mastercam X7 introduced two new predefined strings, `smc_user_dir$` and `smc_mcam_dir$`. These let you access the locations of the Mastercam user folder and the Mastercam installation folder. For X8, we have added `smc_shared_dir$`, which is the location of the Mastercam shared folder. The complete set is as follows:

- `smc_shared_dir$`—location of the **\shared mcamx8** folder
- `smc_user_dir$`—location of the **\my mcamx8** folder
- `smc_mcam_dir$`—location of the Mastercam installation folder

For example, for an X8 user with initials **jrm**, these might be:

- `smc_shared_dir$ = C:\users\public\documents\shared mcamx8\`
- `smc_user_dir$ = C:\Users\jrm\Documents\my mcamx8\`
- `smc_mcam_dir$ = C:\Program Files\mcamX8\ ■`

Enhanced use of `tooltable$`

The `tool table$` variable can now be used as a command word. In connection with this, new values and states have been defined. In addition, a helper variable, `tool table_active$`, is also available. Please read the articles below for full information.

`tooltable$`

The `tool table$` variable enables calls to the `pwrtt$` (and, by extension, `pwrttparam$`) postblock for tooltable processing. When `tool table$` is greater than 0, these postblocks are called during the NCI pre-read process for tooltable processing. Typically `tool table$` is initialized to either **1** or **3** as described below.

Assigning a value to `tooltable$` instead of initializing it

In addition to initializing `tool table$`, you can also assign a value to it in either of these postblocks:

- `pprep$`
- `pq$`

Because these postblocks are executed before MP initially captures the value of `tool table$`, you can set or override its value within them in time for the NCI pre-read process.

This is also true of certain other postblocks, but because they may be called again later, you should not attempt to use them to assign a value to `tool table$`. Please limit this practice to `pprep$` and `pq$`.

Using `tooltable$` as a command word

`tool table$` can also be used as a command word. Do this to initiate tooltable processing from within any postblock.

Note that you should NOT do this from within `pwrtt$` or `pwrttparam$`, since this will create a recursive process.

Even when used as a command word, its value must still be set to **1** or **3** before it is called. Typically you assign a value just before calling it. For example:

```
p_mypostblock
  tool table$ = 3
  tool table$
  tool table$ = -1
```

If `tool table$` is used as a command word at the start-of-file or later, you cannot initialize it to a value; you must assign a value before the `tool table$` command as shown above.

When post processing returns from the tooltable processing, it is important that you set the value of the `tool table$` variable to **-1** to signify that tooltable processing has completed.

The automatic processing and command processing are not mutually exclusive. You can choose to use both methods in your post.

Configuring `tool table$` as a “license plate” value

The `tool table$` value can have an optional second digit in the 10’s place, to the left of the traditional value. This extra digit lets you further configure its processing. Values of **1**, **2**, and **4** are supported.

- **1**—Recall that setting `tool table$ = 3` (or `tool table$: 3`) enables an extra call to `pwrtt$` at the end-of-file NCI code 1003. Setting the 10’s digit to **1** modifies this so that the extra call to `pwrtt$` takes place at the physical end of file. This lets you access comments and other NCI data that occurs after NCI 1003. For example:

```
tool table$ = 13
```

Note that if the post has been enabled for probing, and probe data has been detected in the post, this behavior is automatic. Note also that `gcode$` will be forced to **1003** at the final call to `pwrtt$`. This is to provide backward-compatibility with older posts.

- **2**—When you call `tool table$` as a command word, the normal NCI read process has already progressed to a certain point when the `tool table$` command is encountered. However, when tooltable processing finishes, MP attempts to restore the normal NCI processing at the first operation (start-of-file). Setting the 10’s digit to **2** tells MP to read the NCI back to the point where the `tool table$` command was encountered before resuming normal NCI processing.
- **4**—If tooltable processing ends on the start-of-file and `tl chng_aft$` is active, MP typically does not continue reading the data to the motion line after the toolchange. Setting the 10’s digit to **4** enables this behavior.

Notwithstanding options **2** or **4**, if the `tool table$` command is encountered before the start-of-file, when MP is restoring the read location in the ASCII NCI file after tooltable processing has finished, it will never read past the ASCII NCI location that was current when the `tool table$` command was issued.

This behavior also applies to calls after the start-of-file: rereading the NCI to restore the location at which normal NCI processing will resume will not go past the location in the file from where the tooltable call was initiated.

Note that while tooltable processing is taking place, MP does update the `nci_line$` value.

Values	
-1	Indicates that the tooltable has already been processed. Typically this is set behind-the-scenes by MP but if you are using <code>tool table\$</code> as a command word, you can do this manually.
0	Disable tooltable processing.
1	Call the <code>pwrtt\$</code> postblock at tool changes.
2	Obsolete value. For backward compatibility, if encountered in a modern post, MP processes it the same as 1 .
3	Call the <code>pwrtt\$</code> postblock at tool changes and, in addition, call it again at the end of the NCI file (NCI 1003).
1x	If 1’s digit is 3 , instead of calling <code>pwrtt\$</code> at the 1003 line, it is called at the physical end of file.

Values	
2x	When restoring normal NCI processing, read NCI to the point where <code>tool table\$</code> was called.
4x	When restoring normal NCI processing and <code>tl chng_aft\$</code> is active, read NCI to the motion block after the toolchange.

tooltable_active\$

Indicates whether MP is reading the NCI for tooltable processing or normal processing. For example, during the NCI pre-read for tooltable processing, this will be set to **1**. When reading the NCI for normal post processing, this will be set to **0**. This will also be set to **1** if tooltable processing has been initiated with the `tool table$` command.

Values	
0	Normal NCI processing.
1	NCI read for tooltable processing is active.

Passing arguments to postblocks

Beginning with Mastercam X8, the MP language supports passing values with postblock calls. This means that you can define a postblock with a list of parameters, and then pass arguments to it when you call it from another postblock.

To do this, simply include the list of parameters in the postblock definition. Use parentheses and separate them by commas:

```
# Define postblock with parameters
pca l c _posi ti on( xcoord, ycoord, zcoord, start_angle )
```

If you want, you can break out the parameters on separate lines, but the initial parenthesis must be on the same line as the postblock name:

```
# Define postblock with parameters
pca l c _posi ti on(
    xcoord,
    ycoord,
    zcoord,
    start_angle )
```

Spaces before/after the parentheses and commas are ignored.

The parameters must be numeric variables; strings are not supported. However, you are allowed to use variable names that have not already been declared or initialized. MP will automatically create the variables for you (and initialize them to 0) if they do not already exist.

While you cannot specify an explicit return value for a postblock, you can accomplish the same thing by passing references to variables (as described below). Also, any variables whose values are changed inside the postblock are visible outside the postblock as well.

Once you define the postblock, you can call it and pass values to it; for example:

```
pca l c _posi ti on( xdr1$, ydr1$, zdr1$, 30 )
```

The values that you pass to the postblock can be any valid numeric expression:

- User-defined or predefined numeric variables. This includes `prv_` variables and variables defined as constants.
- Numbers—for example, 30.
- Formulas/functions that return a numeric value—for example, `(abs(xnci $))`. In this case, the formula or function will be evaluated before the postblock is called, and the return value matched to the parameter list. Only formulas and functions that return a numeric value can be used.
- As a general rule, it is a good practice to enclose formula arguments in parentheses.
- Expressions—for example, `3*my_var`.

You cannot skip any arguments when calling the postblock. If you include fewer arguments than are defined in the postblock definition, MP will not generate an error. However, the parameters for which no argument has been supplied will continue to have whatever value they were last assigned.

Conversely, if your postblock call has more arguments than have been defined for the postblock, the extra arguments will be ignored. The typical best practice is for the postwriter to ensure that the number of arguments matches the number of parameters.

You are allowed to use a negative number as an argument; for example, `-30`. However, if you use the minus sign in front of a variable name in your argument list, you must enclose it inside parentheses; for example, `(-yncl $)`.

Passing values by reference; updating argument values

Normally, when you include the name of a variable in a postblock call, only the value of the variable is passed to the postblock. The original variable is left unchanged. For example, in the previous section, when the postblock is called with `pcalc_position(xdr1 $, ydr1 $, zdr1 $, 30)`, the following things happen:

- The current values of `xdr1 $`, `ydr1 $`, and `zdr1 $` are copied to `xcoord`, `ycoord`, and `zcoord`.
- The values of `xcoord`, `ycoord`, and `zcoord` are manipulated by the postblock code.
- The values of `xdr1 $`, `ydr1 $`, and `zdr1 $` are not changed.

You can change this behavior by including the `!` (force update) character in your argument list when calling the postblock. When you do this, the variable in the argument list will be updated when the postblock has finished.

For example, if you include the `!` like this when you call the postblock:

```
pcalc_position( !xdr1 $, !ydr1 $, zdr1 $, 30 )
```

then after `pcalc_position` has finished executing, the values of `xdr1 $` and `ydr1 $` will be updated with the values of `xcoord` and `ycoord`.

Note that variables defined as constants will not be updated, if you pass them as arguments to a postblock.

You are also allowed to include the `!` operator in the postblock definition. However, this is only a convenience to make your post more readable, to specify that the postblock is expecting an argument in this form. It does not have any other effect.

Scope and visibility

Numeric variables that are created in the parameter list of the postblock definition are created and initialized on the second parsing loop in MP. They are initialized to 0 and assigned to format statement 1.

Once they have been created, they can, technically, be used anywhere else in your post. They function just like any other user-defined variable, and their values persist after the postblock has finished executing. However, they cannot be used before the postblock definition. As a best practice though, you should normally use such variables within the postblock, and copy their values to other variables if you want to access them elsewhere.

In other respects, postblocks with parameter definitions can access other variables and their values just like any other postblock.

Previous values

Not only can you use the `prv_` value of a variable as an argument when calling a postblock, you can use it in the parameter list when you define the postblock. When you do this, MP will also automatically create the regular variable when the postblock definition is processed.

The `prv_` states of the argument and the parameter definition do not need to match. For example, if the postblock definition includes a `prv_` in its parameter list, you can still pass it a regular variable as an argument.

Global variables

You are allowed to use global variables both when defining and calling a postblock.

Global variables used in a postblock definition are assigned the value that is passed in when the postblock is called.

Global variables used as an argument when calling a postblock are not solved when the postblock is called. They are assigned a value on return.

Versions of Mastercam before X8

Note that if you use these techniques in versions of Mastercam earlier than X8, MP will not give you an error. However, they have not been tested in earlier versions of Mastercam, and their use is not supported or recommended.

Example

The following example shows how different types of argument values are processed and manipulated when they are passed to a postblock called `pblockvars` that is defined with three parameters.

```
var_1 : 5
var_2 : 10
var_3 : 15

# Postblock definition with list of parameters
pblockvars(var_a, var_b, var_c)
  # Output values passed in as arguments
  "IN CALL -> ", ~var_a, ~var_b, ~var_c, e$
  var_a = var_a + 100, var_b = var_b + 100, var_c = var_c + 100

pheader$

# Output original values
"BEFORE CALL: ", ~var_1, ~var_2, ~var_3, e$

# Call to postblock with arguments
pblockvars(!var_1, 13, (-var_3))

# Output values after called postblock and compare
```

```
"AFTER CALL: ", ~var_1, ~var_2, ~var_3, e$  
"          : ", ~var_a, ~var_b, ~var_c, e$
```

It results in the following output:

```
BEFORE CALL:  var_1 5. var_2 10. var_3 15.  
IN CALL ->  var_a 5. var_b 13. var_c -15.  
AFTER CALL:  var_1 105. var_2 10. var_3 15.  
            :  var_a 105. var_b 113. var_c 85.
```

You can see the effects of passing argument values in three different forms:

- The value of var_1 is passed with the ! character, meaning var_1 is updated by the postblock pblockvars.
- The sign of var_3's value is reversed before it is passed.
- Instead of var_2, a simple number (13) is passed.

NCI changes

New tap_pitch\$ parameter added to 1016 line

A new predefined variable, `tap_pitch$`, has been added for X8. This is output as parameter number 18 on the 1016 NCI line. For tapping tools, this is the pitch of the thread on the tap (distance between threads). This is output in either inches/thread, or mm/thread.

If you use an inch tap in a metric part, the pitch is automatically scaled to mm, and vice versa. In other words, if the tap is 40 threads/inch, the value in the NCI file will be 0.025 for an inch part and 0.635 when used in a metric part.

Posts from earlier versions of Mastercam relied on `n_tap_thds$`, which required logic in your post to properly handle both inch and metric cases. In these cases, using `tap_pitch$` can simplify your post because it is already output in the proper units.

In addition to `tap_pitch$`, one more new parameter will be added to the 1016 line, but this was not completed in time for the Technology Preview. This will show up in a future X8 beta release as parameter 19, and will be documented at that time.

New tool parameters added to 20004 line

Six new parameters (19–24) have been added to the end of the 20004 line:

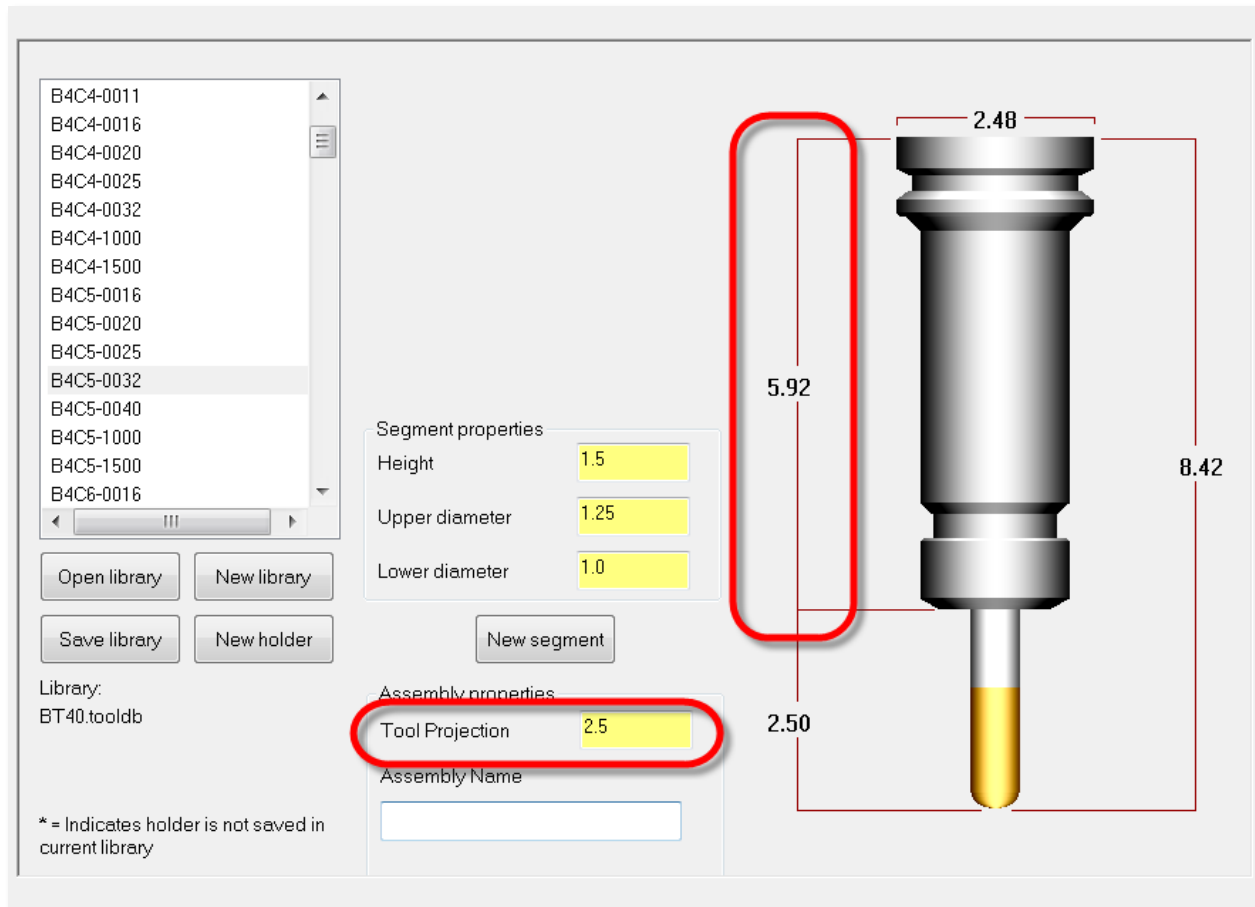
- 19 = chamfer distance
- 20 = drill tip (pilot) diameter (spot/center drills)
- 21 = drill tip (pilot) length (spot/center drills)

The image shows two side-by-side screenshots of Mastercam tool definition dialog boxes. The left dialog is titled 'Define Slot Mill' and the right is 'Define Center Drill'. Both dialogs have a subtitle 'Adjust geometric properties used to define the tool shape.' and are organized into sections with expandable/collapsible arrows. The 'Define Slot Mill' dialog has sections for 'Overall dimensions' (Cutting diameter: 2, Overall length: 2, Thickness: 0.25), 'Tip / corner treatment' (Corner type: Chamfer, Radius: 0, Chamfer distance: 0.0625 - highlighted with a red circle), and 'Non-cutting geometry' (Shank diameter: 0.5). The 'Define Center Drill' dialog has sections for 'Overall dimensions' (Cutting length: 0.3125, Shank diameter: 0.4375, Overall length: 2.75), 'Tip treatment' (Drill diameter: 0.1875, Drill length: 0.1875 - both highlighted with a red circle), 'Drill angle: 118', and 'Shoulder angle: 90'.

- Three additional parameters have also been added (22–24) but are reserved for future use. These will output with zero values until then.

New holder parameters added to 20007 line

Two new holder parameters (12 and 13) have been added to the 20007 line. These represent the stick-out length and the total holder length.



- 12 = **Tool Projection** (stick-out length)
- 13 = **Total holder length**

Note that parameters 6 and 7 from the 20007 line have been deprecated. These were used in X6 and earlier to store the holder diameter and length. These are no longer used by the new tool holders that have been available since X7, but are still output if you are using a tool that was defined in X6 and earlier.

For tool holders created in X7 and later, use parameter 13 to get the holder length. Also use parameter 13 if the tool definition itself was created in X6 or earlier, but you have selected a different tool holder on the **Holder** page in the toolpath settings. In this case, parameters 6 and 7 will continue to have the values of the original tool holder, but parameter 13 will have the length of the holder that is actually being used as represented on the **Holder** page. ■

UpdatePost script tags enhanced for X8

Mastercam X7 introduced a set of tags that you can use to target UpdatePost script logic for specific types or versions of posts. For X8, the tags have been enhanced so that you can target either .PST/.PSM files or .SET files. This lets you create logic in your UpdatePost scripts to target or omit setup sheets during the update process.

To review, the tags tell UpdatePost for which types of posts a particular update function will be executed. These tags come before the function and have two elements, separated by a vertical pipe (|). The first element is the post type, the second is the version: for example, [LATHE|X_4].

```
[LATHE|X_1] [LATHE|X_2] [LATHE|X_3] [LATHE|X_4]
//1 Allow output for comments using gcodes larger than
[POST FILE REPLACE LINE]
1. if gcode = 1008, pbl d, n, pspc, "(", scomm, ")"
2. if gcode >= 1008, pbl d, n, pspc, "(", scomm, ")"
```

To target or omit .SET files, include a second vertical pipe (|) followed by either a SET or PST tag: for example, [LATHE|X_4|PST].

- The PST tag means that the function that follows the tag will only be applied to .PST or .PSM files, not .SET files.
- The SET tag means that the function will only be applied to .SET files.
- If neither tag appears, the function will be applied to both types of files, which is the same functionality as in X7. ■

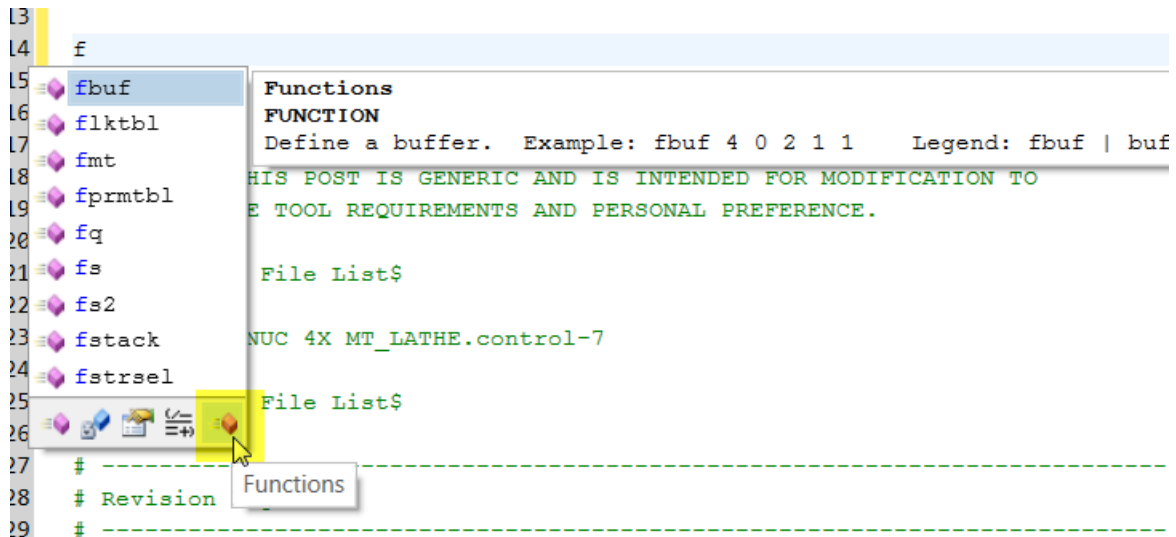
New tool_op\$ codes

There are two new tool_op\$/synctool_op\$ values:

- 72 = lathe contour rough operation
- 154 = SafetyZone linking operation ■

New entries added to AutoComplete

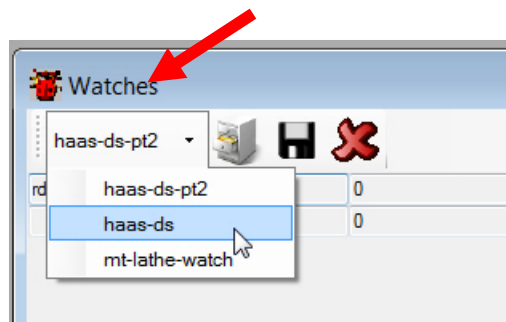
The AutoComplete feature in the Code Expert Editor has been enhanced with additional entries for MP posts. Code Expert now includes information about declaration/definition functions such as `fbuf` and `fstrsel`. If you can never remember which parameter is which in your `fbuf` or `fstrsel` declarations, the new tooltips should prove to be very handy.



Notice that a new button has been added at the bottom of the list. Use it to filter/display the new AutoComplete entries.

Debugger enhancements for X8

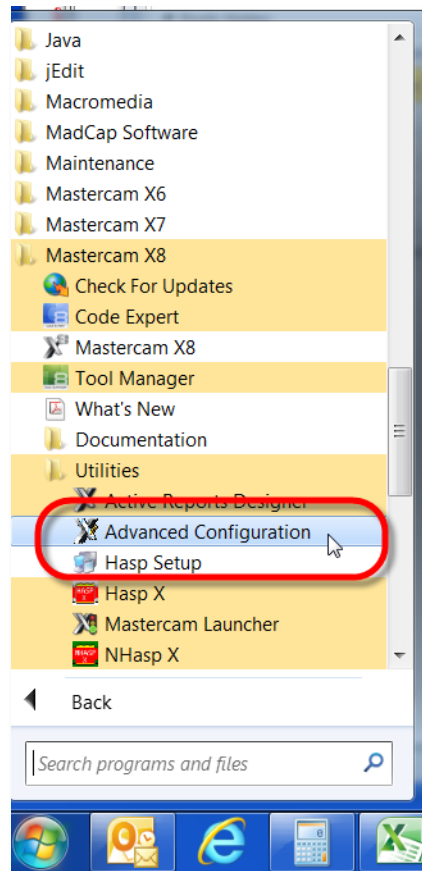
We have enhanced the MP debugger watch list functionality for X8. The **Watches** toolbar now includes a button that you can use to set the folder where the watch list files are stored.




When you select a folder, Mastercam automatically populates the drop list with all the watch list files from that folder. In Mastercam X7, by contrast, saved watch lists could only be stored in the `\Program files\mcamx7` folder.

Enabling the debugger in Windows 8

Most of you know that before you can use the debugger, it needs to be enabled with Mastercam's **Advanced Configuration** utility. This is typically accessed from the Windows **Start** menu:



However, if you are running Windows 8, finding this utility can be a challenge. Unless you have manually configured your interface to make the **Start** button available, the easiest way for most users will be to:

1. Start Windows Explorer with the -E shortcut.
2. Navigate to the Mastercam program folder (**\Program Files\mcamx8**).
3. Run **McamAdvConfig.exe**.

Generic Haas 3X Mill included with X8

Due to popular request from Resellers and users, we have created a generic Haas 3-axis post that will be included with X8. This will correct the spurious A0 output that resulted when you tried to use the current 4-axis post.

In addition to the post, new machine and control definition files have been created. The new post and associated files will be included with the Mastercam product installation.

The new files are:

- **GENERIC HAAS 3X MILL.MMD-8**
- **GENERIC HAAS 3X MILL.CONTROL-8**
- **GENERIC HAAS 3X MILL.PST**